TMAN 5.24-5.88

Taul Manual for Antiballistic Missle Systems

COMMANDS: NOTATION

In the notation here used for commands, the underlining will indicate what parts are compulsory, including spaces. (Extra non-compulsory spaces will be shown.) E.g.,

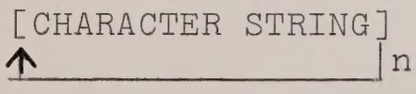
GET TO

LIST represents any monicker being used for a list. Where more than one list is referred to in a command, LIST1, LIST2, LIST3...shall stand for distinct monickers referring to lists. Where a loop format is used (see below), LISTN shall mean the successive monickers used to reference successive lists on successive iterations.

LABEL represents any monicker being used to label a TAUL statement.

PROCNAME represents any monicker being used to name a procedure.

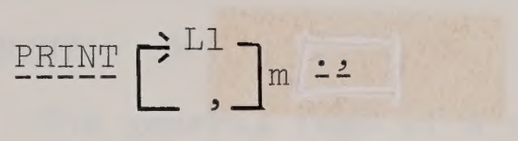
Multiple options are represented by a "loop" arrow, viz.:



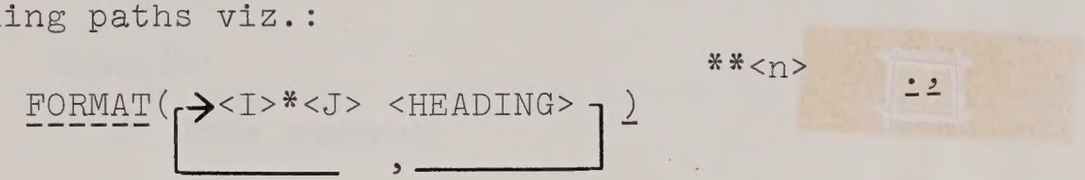
where n represents the number of iterations of this loop possible. This is given as an integer if fixed. If the number is based merely on what fits into the instruction (36 tokens), the subscript will be "wf" (whatever fits). e.g., wf $\leq$ 3. If the number is some large unpredictable figure effectively based upon available storage, the subscript will be "m" (many).

In this loop notation, the arrow should be understood as going directly from the last character of one iteration to the first character of the next iteration. No characters occur during the arrow, except optional spaces or a character specified within the loop.

For example:



If there are optional phrases within a command, they are shown by branching paths viz.:



5C0. MISCELLANEOUS INSTRUCTIONS

COMMENT

The general form of a comment is:

C_ any text that the user desires .,

Example:

```
C  This comment
   can be
   as long as is
   reasonable.,
```

LABEL

Any command may be prefixed by the micro-command

LABEL ..

which may be any monicker used as a label. It must be followed by two periods (the lazy-colon).

The space between label and lazy-colon is optional, as is the space after the lazy-colon.

A label defines a possible destination for a GO TO statement. It must be the first token of the TAUL statement it labels. Thus only one label is possible in a given command; the processor will not recognize a second label.

Except for having its possible length shortened by two tokens, a command prefixed by a label is otherwise unchanged.

Labelling a statement is distinct from addressing that label with a GO TO. A label may have many, one, or no GO TO statements pointing to it.

If more than one statement has the same label, TAUL will invariably branch to the first of them.

Any command whatever may have a label. This includes the two commands following an IF.

Digitized by the Internet Archive
in 2025

https://archive.org/details/tmanforantiballi00unse_0

5C1. DATA INSTRUCTIONS

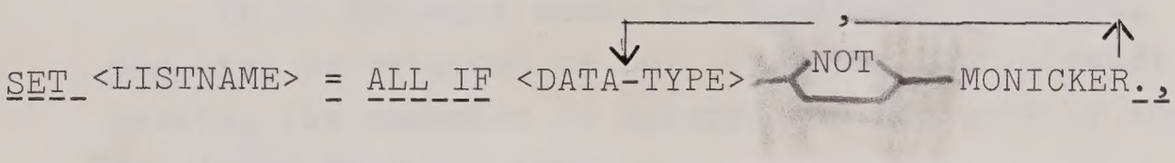
SET

The SET command is the basic command for retrieving and replacing data in TAUL. It can be broken down into four subcommands:

1) SET - ALL IF
2) SET - OF
3) SET =
4) SET * SYNONYM

1. The SET - ALL IF command.

This command is the basic data creating command in TAUL. Its format is:



Examples are:

a) SET L1 = ALL IF DPS GEN NR D03S01A, DOC TYPE NOT PL.,
b) SET L1 = ALL IF APPLICABLE SITES WH.,

This command will set up a list with a group of file pointers (AIDS numbers) to all the files in the data base that satisfy the criteria of the command. These elements can be easily converted to a pointer to specific data within these files as will be shown a little later.

<DATA-TYPE> can be any of the following:

DOC TYPE
SECURITY OF DESIGN
DPS GEN NR

APPLICABLE SITES
ISSUE DATE
DOC DATE
NAME
DEPT
REVNR

DPS GEN NR will retrieve all documents from the specified level on down to the lowest level. For example:

SET L1 = ALL IF DPS GEN NR D03S.,

will retrieve D03S as well as D03S01A02B03C02 but not D03.

If no document meets the specified criteria, the list is returned as null. If the list already exists, its contents is deleted and replaced by the retrieved file pointers.

2. SET - OF command.

Once a list of file pointers is created, it is possible to create a list of items pointing to certain attributes of these files. This operation is done by the SET - data OF command.

Its format is:

SET <LIST NAME1> = <ATTRIBUTE>OF ——— TITLE / * SYNONYM <LISTNAME2> ——— .,

Examples are:

SET L2 = TITLE OF L1.,
SET LR = HEADING OF * S1A GOOP.,
SET ZT = DESCRIPTION OF FORNI.PREPROCESSOR.,
SET LIST1 = PURPOSE OF L1.,

This command will create a list, <LISTNAME1>, that points to an attribute within the file pointers specified by <LISTNAME2> or the title or synonym of the file desired. If used, <LISTNAME2> must exist before the command is executed and must have been formed by a SET - ALL IF command or a SET*SYNONYM.

<ATTRIBUTE> can be any of the following:
any subfile name - this is a special case of this command
TITLE
SYNONYM
NAME (person responsible)
DPS GEN NR
DOC TYPE
SECURITY OF DESIGN
APPLICABLE SITES
PRELIM DATE
DOC DATE

Again all spaces shown above are only one space - no more and no less. If <LISTNAME1> exists, its contents is deleted and it is reformed.

3. SET = command

This command copies lists. Its format is:

SET <LIST1> = <LIST2>.,

Examples are:

SET APPLES = PAIRS.,
SET NIKE.X = NULL.,

The command takes each element on <LIST2> and places it on <LIST1>. If <LIST1> exists prior to this command, its contents are deleted and <LIST2> is copied on to it. <LIST2> will remain unchanged.

4. SET*SYNONYM command

This command will create a list that is one element long. The element will be equal to the file pointer of the file specified by the synonym. The format is:

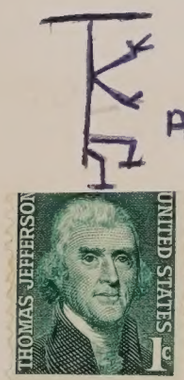
SET <LISTNAME> = * <SYNONYM>.,

Example:

SET AIDSNO = * S1AGETUP.,

This will create a single element list - AIDSNO - containing the file pointer (AIDS number) of the document with the synonym S1AGETUP.

This concludes the discussion of the SET commands.



PAUL

STAMPS OUT LETHARGY IN LIST PROCESSING

JAM

LIST

The LIST command is similar to the set command, but it goes one step further to provide instant output. The format is the same as the SET command but in every case the phrase SET <LISTNAME> is replaced by LIST. For example:

a) LIST ALL IF DPS GEN NR D03S01.,

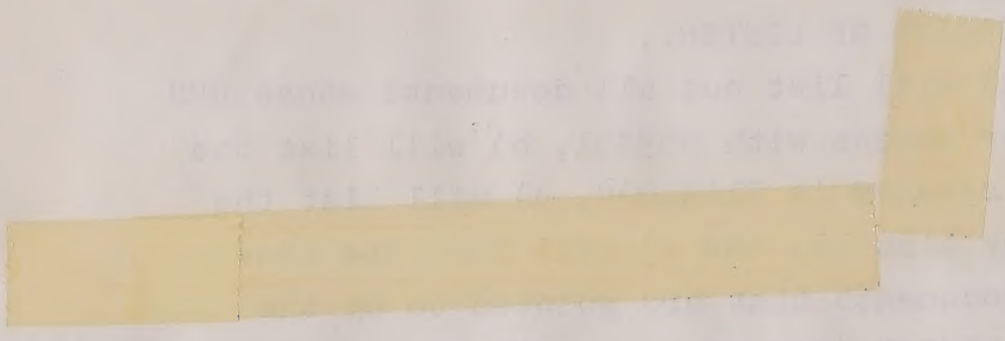
b) LIST * S1ABADOC.,

c) LIST L1.,

d) LIST HEADING OF LISTER.,

The example in a) will list out all documents whose DPS generation number begins with D03S01, b) will list the documents whose synonym is S1ABADOC, c) will list the data specified by List L1, and d) will list the heading subfile of the documents that are pointed to by the items on a list named LISTER (formed by an ALL IF or a *SYNONYM).

When using the LIST..OF option, the attribute must be a subfile name. Any other attribute is ILLEGAL.



REPORT

This command controls the format of the output of the list command. In coding sequence, it may appear anywhere before a list command. If no REPORT command has been issued, then the standard options are taken.

The format is:

REPORT ↳<OPTION>, .,

The legal options are:

a) STANDARD - this resets all options to the standard form which is single spacing, no top of form, output on high speed printer if GECOS, TTY if Time Sharing, no line numbers, 72 characters per line, 50 lines per page.

b) LINE NUMBERS - This will cause line numbers to be printed before each line. The count is initialized at the beginning of each subfile.

c) n CHARACTERS PER LINE - This will print n characters per line where 1<n<120.

d) PRINTER - o/p on high speed printer

e) PUNCH - o/p on punched cards

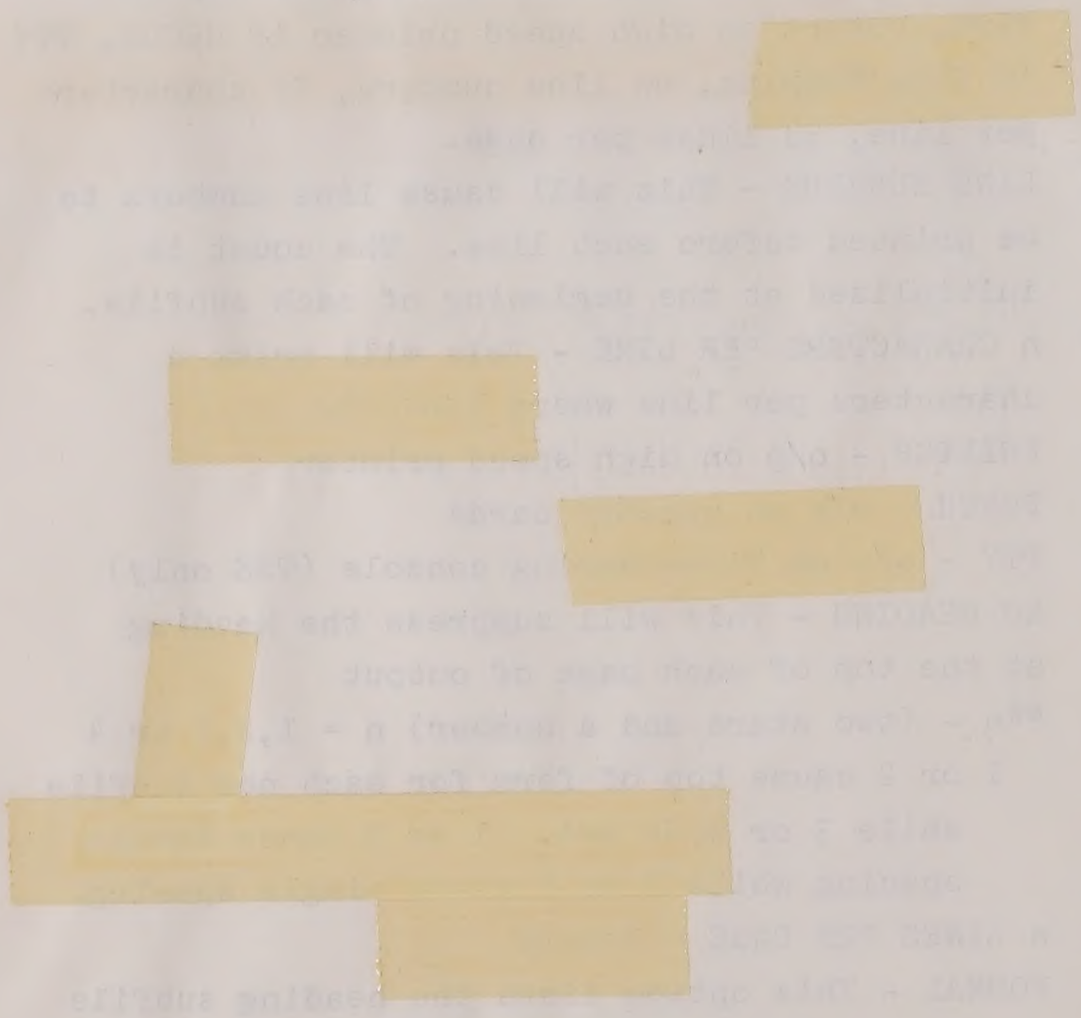
f) TTY - o/p on Time-Sharing console (TSS only)

g) NO HEADING - This will suppress the heading at the top of each page of output

h) **n - (two stars and a number) n = 1,2,3 or 4
1 or 2 cause top of form for each new subfile while 3 or 4 do not. 1 or 3 cause double spacing while 2 or 4 cause single spacing.

i) n LINES PER PAGE - 1<n<50

j) FORMAL - This option lists the heading subfile as a cover sheet and table of contents



Examples of the REPORT command are:

a) REPORT FORMAL.,
b) REPORT PRINTER, 80 LINES PER PAGE, **4.,
c) REPORT STANDARD.,

Remember that the LIST command will not reset the output format by itself. It will use the last setting of the options. Therefore, if the user writes "REPORT PUNCH.," as his first command, all output will come out on the card punch unless the user terminates this option with a "REPORT PRINTER.," a "REPORT TTY.,", a "REPORT STANDARD.,", or a "REPORT.,". (The Command "REPORT.," has the same effect as the command "REPORT STANDARD.,".)

PRINT, FORMAT

The PRINT and FORMAT commands are used to produce columnar output from lists created by the SET-OF command that use any attribute EXCEPT subfile names. These commands cover the lists that cannot be processed by the LIST command.

The format of the PRINT command is simply:

PRINT [LIST, N] m ;

The format of the FORMAT command is:

FORMAT ([<I>*<J> <HEADER>,] K) **<n> ;

where:

I = a number specifying the column where the output should start

J = The width of the field

K = Parameter m in the list command. In other words, this phrase must appear once for every list specified in the PRINT command.

n = 1 or 3 for double spacing, 2 or 4 for single spacing
1 or 2 for top of form, 3 or 4 for no top of form

HEADER = a monicker which will appear at top of form as a heading.

If any element is longer than the specified field width, then the remainder of the characters will appear on the succeeding lines.

The actual output will occur when the FORMAT command is found. The most recent PRINT statement will be used, but usually it is good programming practice to write the two statements consecutively. When the PRINT command is issued, the lists specified must exist. Examples are:

a) PRINT L1, L2, L3.,
FORMAT (1*20 TITLES, 21*15 DPS.NO, 39*10
PERSON.RESPONSIBLE) **3.,

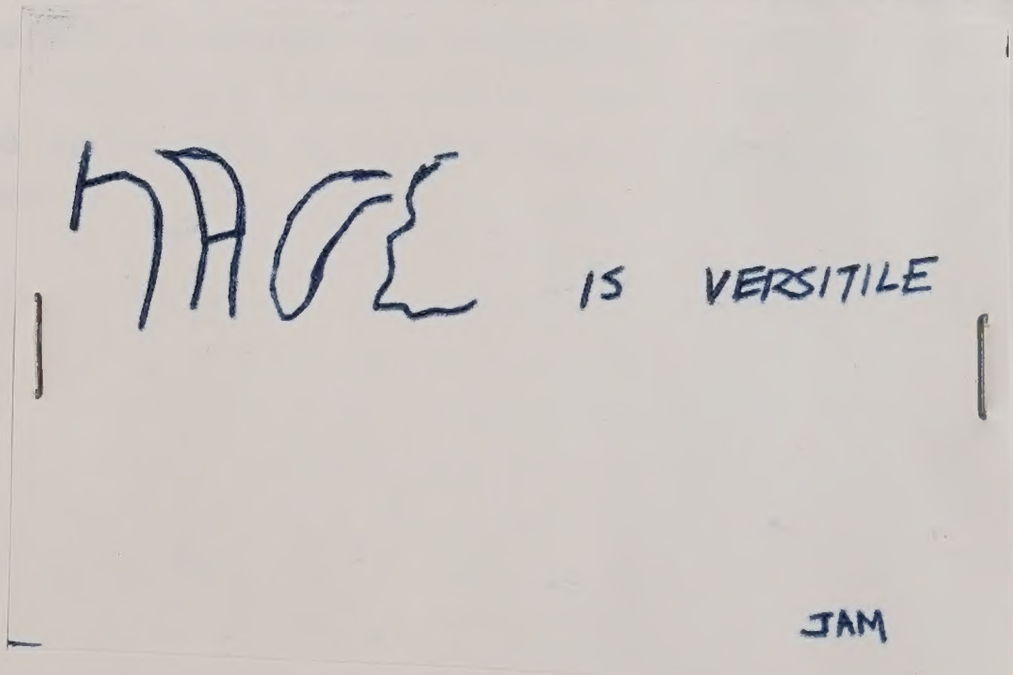
in the above example PERSON.RESPONSIBLE will come out on 2 lines.

b) PRINT LISTER.,
FORMAT (40*25 OUTPUT).,

The default value for <n> is 3, that is double spacing and no top of form.

c) PRINT OUT, INTO.,
FORMAT (1*20 O.P, 30*25 ==DATA TO OUTPUT==).,

Notice literals can be used to get spaces into headings.



SORT ALPH

The SORT ALPH command is used to sort lists. The command has two formats:

a) SORT ALPH LIST.TO.BE.SORTED.,

b) SORT ALPH <LIST1> CARRYING [,<LISTN>]m .,

Format a) is used simply to sort the elements on the list according to the ASCII collating sequence. Format b) is used to arrange the lists to be carried in the same order as the list being sorted. An example of coding that makes use of the CARRYING option is:

```
SET L1 = ALL IF DPS GEN NR DO3.,
SET L2 = DPS GEN NR OF L1.,
SORT ALPH L2 CARRYING L1.,
SET L3 = TITLE OF L1.,
```

Line 3 will sort L2 and will also arrange the file pointers in L1 in order of dps generation number. Line 4 will create a list, L3, equal to the titles of the documents specified in L1 and in order of dps number.

LIMITATIONS!

The list being sorted must have been formed using the SET-OF command and NOT using a subfile name.

The lists being carried can be created in any manner and do not have to be the same length as the list being sorted.

LOGICAL COMBINATIONS OF LISTS

There exist four commands which produce logical combinations of lists, setting a new list equal to some Boolean function of two other lists. The functions are, with some qualifications, union (UNION), intersection (INTS), difference (DIF), and symmetrical difference (SYMDIF). We will consider each in turn.

(The shaded portions of the circles below are intended to make the particular Boolean functions visually understandable.)

The format of all four of these commands is identical, viz.:

LIST1 = LIST2 [LOGICALOP] LIST3 .,

The different operators will be specified, and examples given, below.

UNION

The UNION command creates a list which is the union, or logical sum, of two lists.

The format of this command is

LIST1 = LIST2 UNION LIST3 .,

LIST1, LIST2 and LIST3 may be any lists. Any two, or all three, may be the same lists.

The result will be as follows: LIST1 will become a list whose contents are the contents of LIST2 and the contents of LIST3. Any elements that are common to LIST2 and LIST3 will appear once.

If there is already a LIST1 (different from LIST2 and LIST3), its old contents are clobbered and lost. If LIST1 does not exist, it is created. If LIST2 or LIST3 do not exist, an error condition results.

Examples:

```
CHILD = MOTHER UNION FATHER.,
IRVING = 72S UNION 360.,
2B    = IRVING UNION IRVING.,
```

INTS (Intersection)

The INTS command creates a list which is the intersection, or logical conjunction, of two lists.

The format of this command is

LIST1 = LIST2 INTS LIST3 .,

LIST1, LIST2 and LIST3 may be any lists. Any two, or all three, may be the same lists.

The result will be as follows: LIST1 will become a list whose contents are any elements which are on both LIST2 and LIST3.

If there is already a LIST1 (different from LIST2 and LIST3), its old contents are clobbered and lost. If LIST1 does not exist, it is created. If LIST2 or LIST3 do not exist, an error condition results.

Examples:

```
CONTESTED = REPUBLICAN INTS DEMOCRAT.,
ZIG = ZIG INTS ZAG.,
NIKE.X.RADAR = NIKE INTS RADAR.,
```

DIF (Difference)

The DIF command creates a list which is the difference, or one-sided logical subtraction, between two lists.

The format of this command is

LIST1 = LIST2 DIF LIST3 .,

LIST1, LIST2 and LIST3 may be any lists. Any two, or all three, may be the same lists.

The result will be as follows: LIST1 will become a list whose contents are the contents of LIST2 except for any elements which are also on LIST3.

If there is already a LIST1 (different from LIST2 and LIST3), its old contents are clobbered and lost. If LIST2 or LIST3 do not exist, an error condition results.

Examples:

```
YUMYUM = PITTISING DIF PEEPBO.,
C = A DIF B.,
BOOP = BOP DIF BOOP.,
```

(Note that in this last example, the new BOOP has nothing in common with the old.)

SYMDIF (Symmetric Difference)

The SYMDIF command creates a list which is the symmetric difference, or two-sided logical subtraction, between two lists.

The format of this command is

LIST1 = LIST2 SYMDIF LIST3 .,

LIST1, LIST2 and LIST3 may be any lists. Any two, or all three, may be the same lists.

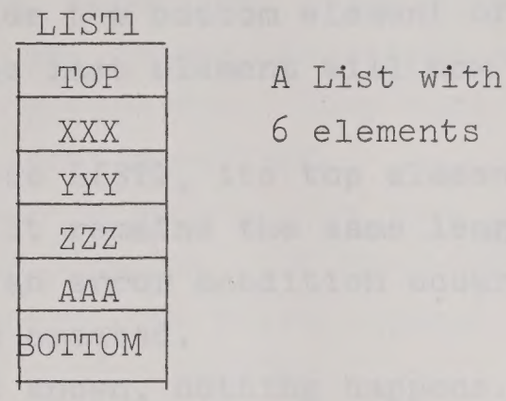
The result will be as follows: LIST1 will become a list whose contents are the contents of LIST2 and LIST3 except for any elements which are on both.

If there is already a LIST1 (different from LIST2 and LIST3), its old contents are clobbered and lost. If LIST1 does not exist, it is created. If LIST2 or LIST3 do not exist, an error condition results.

AUL REACHES NEW

POP

We may think of a list as having a top and bottom, viz.:



The POP instruction involves two lists: the one which is popped, and the one which is popped to.

Popping a list means removing its top element. Popping a list to another list means that the removed top element of the first list goes to the bottom of the second list.

In TAUL's POP instruction, both LIST1 and LIST2 must be specified. The format is:

POP LIST1 TO LIST2 .,

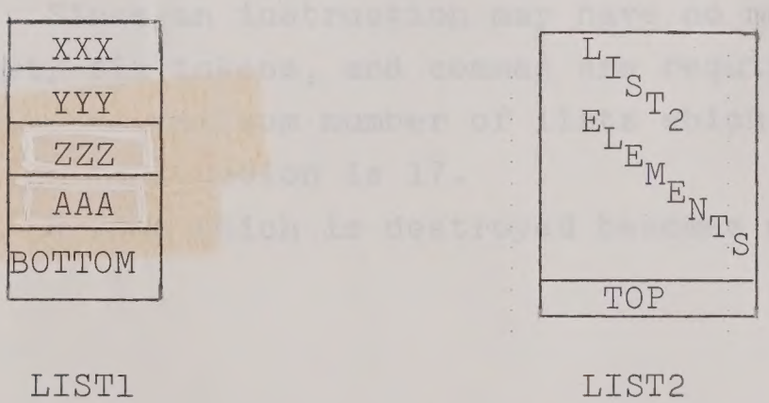


Figure XX: LIST1 and LIST2 after command

The result will be as follows: the first item of LIST1 will have been removed, so that the second item has become first; the item removed from the top of LIST1 will have been made the bottom element of LIST2, and LIST2's previous last element will now be next-to-last.

If LIST1 is the same as LIST2, its top element is popped to its bottom, and it remains the same length.

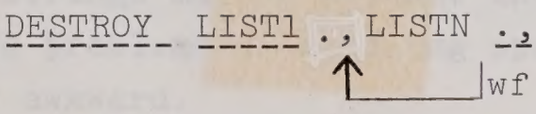
If LIST1 is unknown, an error condition occurs. If LIST2 is unknown, it is created.

If LIST1 is empty but known, nothing happens.

DESTROY

The DESTROY command destroys one or more lists, removing the list pointers themselves from core and removing any associated strings from core. The space thus cleared is returned to the pools of free space.

The format is:



(The comma is necessary if there is more than one list.)

Since an instruction may have no more than thirty-six tokens, and commas are required after each list, the maximum number of lists which may be destroyed in one instruction is 17.

A list which is destroyed becomes unknown.

STRING OPERATIONS (sometimes called "Star Functions")

A number of string operations are available in TAUL. These permit copying, combining, dismantling and measuring textual strings. These operations are intended for lists of string pointers. They work on the first elements of the lists specified.

The star functions operate on any list that contains strings or descriptor pointers. If the list is made up of descriptor pointers, the first element of the list is converted to a string pointer.

Star functions alter only the lists named to the left of the equals-sign. Lists enumerated to the right of the equals-sign are not affected.

Note that these operations usually take place by copying string pointers, rather than by moving or copying strings themselves. Thus different pointers to the same strings may be brought into being. This in turn makes problems of handling space and discarding data more awkward.

ALPH (sometimes called "Star Alph")

This command creates a list with one or more elements.

The format is:

```
* LIST = ALPH (ANYTEXT)  .,
```

where ANYTEXT is any text literal, including numbers if desired.

The alphabetic text ANYTEXT is made into one or more tokens according to a variant of the usual space convention.

The token (or tokens) are made the first (and second, etc.) elements of LIST. If LIST already exists, its previous contents are clobbered and lost.

(Note: This is an exception to the rule whereby string commands operate on the first element of the list specified.)

If LIST does not exist, it is created.

Examples:

```
* JOE  =  ALPH  (A)  .,
* PALINDROME=ALPH(ABLE WAS I ERE I SAW ELBA).,
```

In this example the list PALINDROME has seven elements.

```
*NUMERICAL.CONSTANT = ALPH (792.684)
```

In the last example we see the use of ALPH to put a number in. This is the principal means of defining program constants, etc. (for counting loops, words, etc.). It is also a principal means of getting test data, for debugging purposes, into storage where they can be used.

If ANYTEXT is only one alphameric word with no space, it becomes the first element of the list named. If it is a series of words, they will become separate elements.

Each set of characters set off by spaces will become a separate token. For example,

BOOP BOOP A DOOP

will be made into four tokens.

Since this command uses the conventional tokenizing mechanism, spaces are ordinarily lost, taken to signal the beginnings of new words and therefore tokens. But it is possible to override the tokenizing mechanism by using literal marks. For instance,

==HI DE HO==

will put HI DE HO, with two spaces, all into one token.

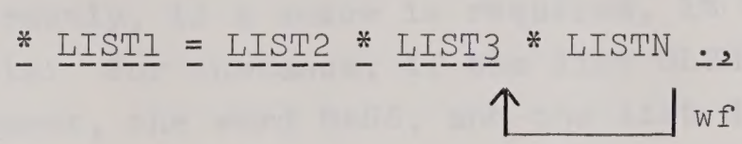
One strange thing about the data created by the ALPH command is that it can be both LISTed or PRINT-FORMATed.

CONCATENATE

This instruction sets LIST1 equal to a single element, a string created from the top elements of LIST2 and LIST3 (and the LISTNs, if any).

Note that if LIST2 has only one element, the IMAGE command may be considered a trivial case of the CONCATENATE command.

The format is:



If LIST1 does not exist, it is created; otherwise its contents are clobbered and lost. If LIST2 or LIST3 does not exist, an error message will result.

If the top element of LIST2, ...LISTN happens to be not a string pointer but a file pointer, an error message will result.

Example:

Suppose you have a list A1 whose top element is the string WHIZ and another list B2 whose top element is the string BANG. Then the instruction

```
*Z = A1*B2.,
```

sets list Z equal to only one element, a string WHIZBANG.

(Note that since the system will concatenate the strings directly, if a space is required, it will have to be put in. For instance, if the list GLUMP consists of one element, the word MANS, and the list GLOP consists of one element, the word LAUGHTER,

```
*Z = GLUMP*GLOP.,
```

will create the combined string MANSLAUGHTER. But the routine

```
*SPACE=ALPH (== ==).,
*Z=GLUMP*SPACE*GLOP.,
```

will set Z equal to the string MANS LAUGHTER.

SUBSTRING

This operation takes a substring of a string, which must be the first item of a specified list.

The format is:

* LIST1 = SUB (LIST2 , LON1 , LON2) .,

LIST1 and LIST2 must be lists. LON1 and LON2 may be either a List or Number (decimal integer).

The contents of the first element of LIST2 are copied into a new string, which becomes the sole contents of LIST1. The copying takes place from the character specified by LON1 for the number of characters specified by LON2. If LON2 is not specified, copying continues to the end of the string. The contents of LIST2 are unaffected.

LIST1 may be any list. If it already exists, its previous contents are clobbered and lost. If it does not exist, it is created. The top element of LIST2 may be a string pointer or a file pointer. LON1 and LON2 may be lists whose first element is a decimal integer, or they may simply be decimal integers to be used literally.

If LIST2 does not exist, an error results. If LON1 or LON2 are not and do not yield decimal integers, an error results. If LON1 or LON2 are too large for the particular contents of the first item of LIST2, they will be treated as if they point to the end of the string. If LON2 is larger than LON1, a null string will result.

Examples:

Suppose the top item of VILLAINS contains the string NECROMANCER. Then the command

*ABSTRACTION = SUB (VILLAINS, 4, 7).,

would deposit in the sole element of the list ABSTRACTION the string ROMANCE.

DELETE

The DELETE instruction finds a specified string in another string, removes it, and closes the gap.

The format is:

* LIST1 = DELETE (LIST2 , LIST3) .,

This instruction removes any occurrence of the string in the first item of LIST3 from the first item of LIST2, and stores the result in LIST1.

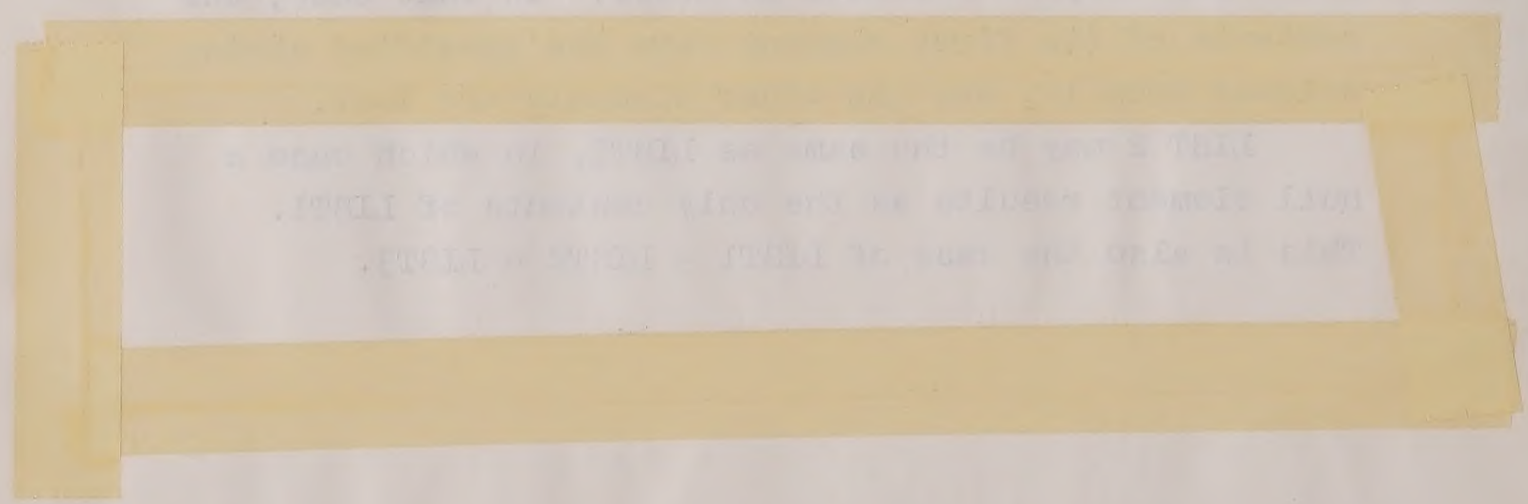
LIST1 may be any list. If it already exists, its current contents are clobbered and lost. If it does not exist, it is created. LIST2 may have as its first element either a string or a file pointer. LIST3 must have a string as its first element.

If the string specified by LIST2 has no occurrences of the string specified by LIST3, nothing happens, except that the first element of LIST2 is copied into LIST1. (Check this with Harold.) Thus the instruction may be used to shorten strings in particular ways.

If LIST2 does not exist, an error occurs. If LIST3 does not exist, an error occurs. If LIST3's top element is not a string pointer, an error occurs. (Check with Harold: could it be a literal?)

LIST1 may be the same as LIST2. In that case, the contents of its first element have the specified string deleted from it, and its other elements are lost.

LIST 2 may be the same as LIST3, in which case a null element results as the only contents of LIST1. This is also the case of LIST1 = LIST2 = LIST3.



Examples:

Suppose LIST2 specifies the string SUSQUEHANNA and LIST 3 merely specifies the letter U. Then

```
* Z = DELETE (LIST2, LIST3).,
```

leaves Z containing SSQEHANNA.

Suppose POOHEM contains the text

```
THE MORE IT SNOWS, TIDDLEY POM,
THE MORE IT GOES, TIDDLEY POM,
THE MORE IT GOES, TIDDLEY POM,
          ON SNOWING.,
AND NOBODY KNOWS, TIDDLEY POM,
HOW COLD MY TOES, TIDDLEY POM,
HOW COLD MY TOES, TIDDLEY POM,
        ARE GROWING.
```

Suppose also that TIDDLEY.POM contains the text

```
TIDDLEY POM
```

In that case the instruction

```
*BASIC = DELETE (POOHEM, TIDDLEY.POM).,
```

would yield a BASIC whose single element contained

```
THE MORE IT SNOWS, ,
THE MORE IT GOES, ,
THE MORE IT GOES, ,
      ON SNOWING.,
AND NOBODY KNOWS, ,
HOW COLD MY TOES, ,
HOW COLD MY TOES, ,
     ARE GROWING.
```

Suppose the list NEWS contains the text

```
THE PRICE OF RIB ROAST HAS BEEN
    REDUCED TO 59¢ A POUND,
    TOMORROW ONLY.
```

And suppose further that the list TEST.DATA contains the word

```
SCHISTOSOMIASIS
```

In that case the instruction

```
*TWEET = DELETE (NEWS, TEST.DATA).,
```

would have no effect whatever, save that NEWS would be copied into TWEET.

Note that the string specified by LIST3 may contain spaces

Even if a word is removed from text in a subfile, only one space will replace it. Example: suppose LIST2 is a subfile pointer pointing to

THE TROUBLE WITH THE SUBSYSTEM...

and we order

* CHUG = DELETE (LIST2, T).,

where T is a list whose first element is THE. The text that will result is

TROUBLE WITH SUBSYSTEM ...

This compression of spaces is handled in much the same way that it is within the tokenizing mechanism.

We may also think of DELETE as getting two substrings, and joining them together. Or the DELETE instruction might be thought of as the opposite of the SUBSTRING instruction. While SUBSTRING takes a string from the middle or end of another string and throws away the rest, DELETE throws a string away from the middle of another string and closes the gap accordingly.

However, DELETE operates not by numbers of character positions within the string, but with the actual image of the string to be deleted.

Note that DELETE is quite distinct from the other data-removing operations, DESTROY and SET ... = NULL. (See p.). DELETE operates within a single list item, DESTROY and NULL operate upon entire lists.

INDEX

This instruction finds a particular string in another string, and takes note of where it is in that string.

The format is

* LIST1 = IN (LIST2 , LIST3) .,

The string specified by the first element of LIST3 is sought in LIST2. When it is found, its location (in character positions) is stored in LIST1.

Note that line feed codes do not count.

LIST1 may be any list. If it already exists its contents are clobbered and lost. If it does not exist, it is created. The top element of LIST2 may be a string pointer or a file pointer. The top element of LIST3 must be a string, otherwise an error will occur.

If the string specified by the top element of LIST3 does not occur in LIST2, the resulting value stored in LIST1 will be zero.

Examples:

```
* F = IN (LYNDABIRD, GEORGE).,
* N = IN (M,O).,
```

The feasibility and results of these commands depend on the contents of the lists. For instance, in the last example, if M contains

HOW NOW BROWN COW

and O contains OW, the resulting value of N will be 2.

Note that text may be scanned for particular words from left to right, using alternating INDEX and SUB instructions. Example:

Suppose we wanted to scan text in some new computer language looking for the command word SHOVE-- and performing operations depending on the position of this command word.

Suppose the list INPUT contains as its first item an input string; suppose further that list S contains the word SHOVE. Then the following program would find the respective positions of the words.
(NUMBUF is a list we will use to hold results temporarily. Z contains the number zero. OCCURRENCES will be a list telling where the word SHOVE occurs on the input line.)

```
        NOTHER.. *NUMBUF = IN (INPUT,S).,
        IF NUMBUF EQ Z., GO TO OVER.,
        *INPUT = SUB (INPUT,1,NUMBUF).,
        POP NUMBUF TO OCCURRENCES.,
        GO TO NOTHER.,
        OVER..
```

LENGTH

This instruction ascertains the length of a string. The format is

```
* LIST1 = LENGTH ( LIST2 ) .,
```

The length of a string, the first item of LIST2, is made the contents of LIST1.

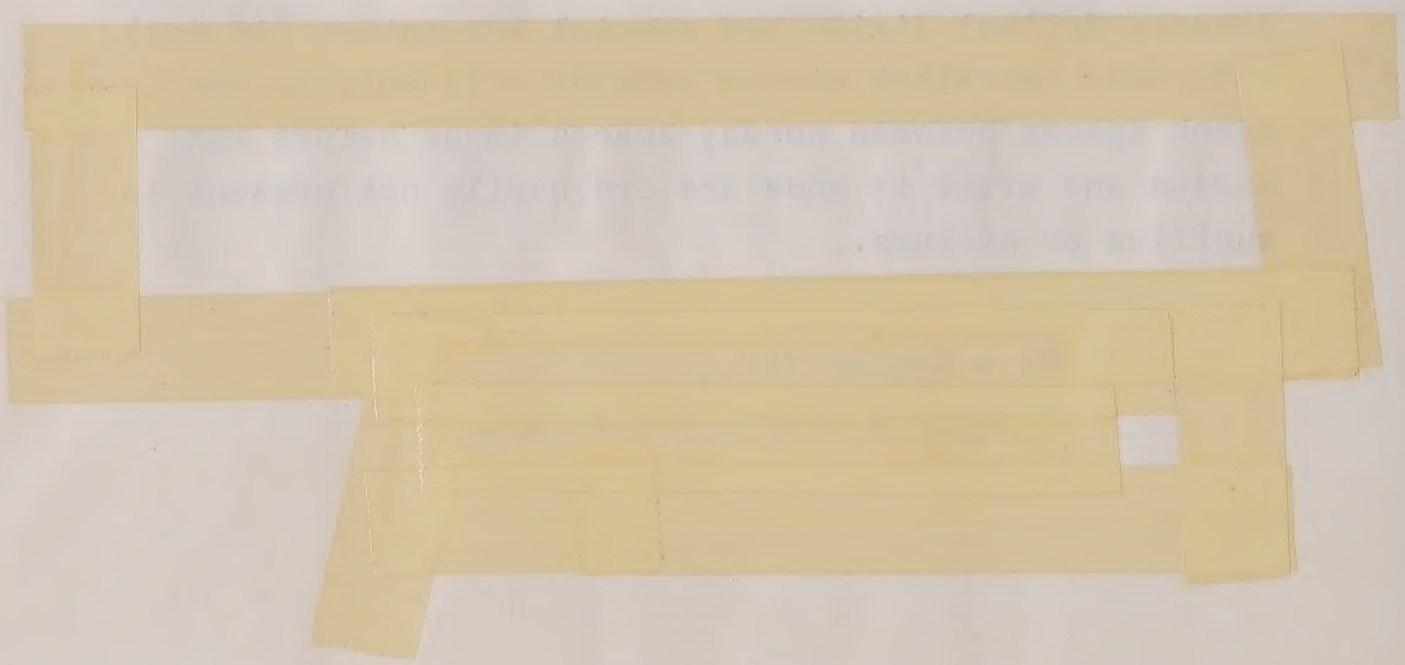
LIST1 may be any list. If it already exists, its contents are clobbered and lost. If it does not exist, it is created.

LIST2 may have as its first element a string or a file pointer. If LIST2 does not exist, an error occurs.

The length is in number of characters, including spaces, decimal digits and special characters (if any). Note that the blank spaces counted will only be the blank spaces between words; spaces input before text begins and after it ends are ordinarily not present in subfiles or strings.

Examples:

```
*A = LENGTH (B).,
*CUBITS = LENGTH (PYRAMID).,
```

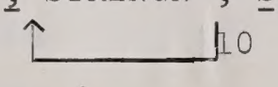


GET TO

This instruction extracts textual strings from between exactly specified delimiters (boundary text strings). It performs this extraction from entire lists with one instruction, rather than operating on the first element alone, as the star functions do. Essentially, it was designed for pulling comparable text units out of equivalent contexts over a number of similarly formatted texts-- the AIDS files.

The format is

GET TO LIST1 LIST2 FROM STRING1 , STRINGN , STRING2 QUANTITY .,



LIST1 is the name of the list that these data must be gotton to. LIST2 is the list they will be found on and extracted from. STRING1 is the left delimiter of the desired text section appearing in literal form, STRING2 is the right delimiter, appearing in literal form. The STRINGNs, if any, are strings that must be contained within the text sections sought. (we will refer to this as "contained material.") QUANTITY may be an integer-- the maximum number of sections desired from any one list element of LIST2-- or the word ALL, meaning all the sections meeting the specifications.

If LIST1 already exists, its contents are clobbered and lost. If it does not exist, it is created. LIST1 will become a list of strings, regardless of the list type of LIST2.

LIST2 must exist, or an error results. LIST2 may be a list of either file pointers or strings, or a mixed list containing both types. (Note that every item of LIST2 is operated upon.) The STRINGs may be any single characters or multi-character tokens; however, if they are to contain spaces, they must be surrounded by literal marks (==LIKE THIS==). Presence of the STRINGNs are checked with respect to their sequence in the command.

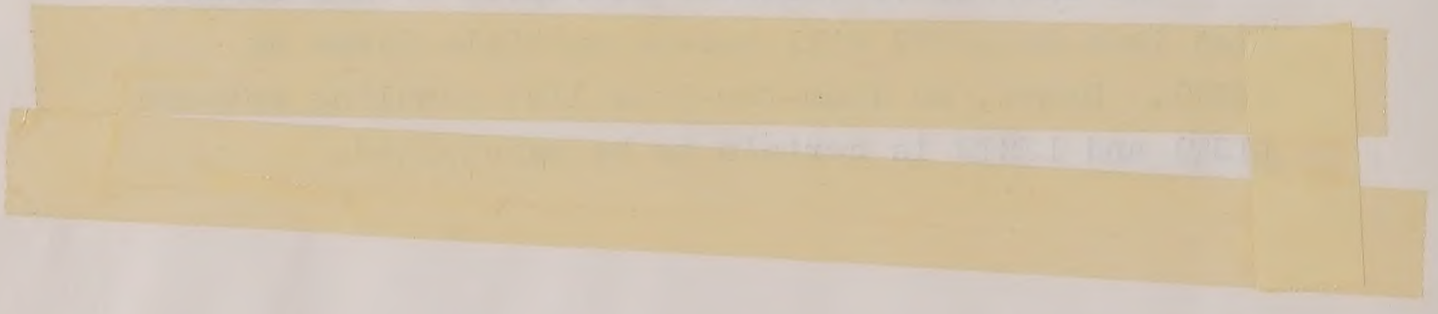
The delimiters themselves do not appear in the strings returned. What you get is

DELIMITER1 ... DELIMITER2

If for the sake of readability you want the delimiters in, they need to be added by means of other string operations.

If a list item contains no section as requested no harm is done, and GET TO passes back an asterisk.

The successive elements extracted from a single list item on LIST2 will become multiple items on LIST1. Hence, no item-for-item list coupling between LIST1 and LIST2 is certain to be maintained.



Examples:

For example, if EXERCISES is a list containing the file and subfile number for a document having the following words

THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG. THE QUICK BROWN FOX JUMPED OVER THE LAZY CAT. A QUICK BROWN FOX LEAPED OVER THE LAZY DOG.

it could be manipulated by the following command:

GET TO FOXUP EXERCISES FROM QUICK,FOX,DOG ALL.,

In this case, the delimiters are QUICK and FOX, and the contained material is DOG; so looking at EXERCISES, it can be seen that pointers will be set up in FOXUP pointing to

BROWN FOX JUMPED OVER THE LAZY

and

BROWN FOX LEAPED OVER THE LAZY

In another example, if LIST2 contained the following entries (considering all characters to be literally present):

"*MARY HAD A LITTLE LAMB/"*ITS FLEECE WAS WHITE AS SNOW.
"*AND EVERYWHERE THAT MARY WENT/THE LAMB WAS SURE TO GO/
IT FOLLOWED HER TO SCHOOL ONE DAY/"*IT WAS AGAINST THE RULES/IT MADE THE CHILDREN LAUGH AND PLAY"*TO SEE A LAMB IN SCHOOL/

,it could be manipulated by the following command:

```
GET TO LAMBY.PI LIST2 FROM "*,/ ALL;
```

In this case, the delimiters are "* and/; so looking at LIST2, it can be seen that the respective elements of LAMBY.PI will be (numbered here for clarity):

1. MARY HAD A LITTLE LAMB
2. AND EVERYWHERE THAT MARY WENT
3. IT WAS AGAINST THE RULES
4. TO SEE A LAMB IN SCHOOL

A more AIDSlike example would be the following. In certain subfiles the string

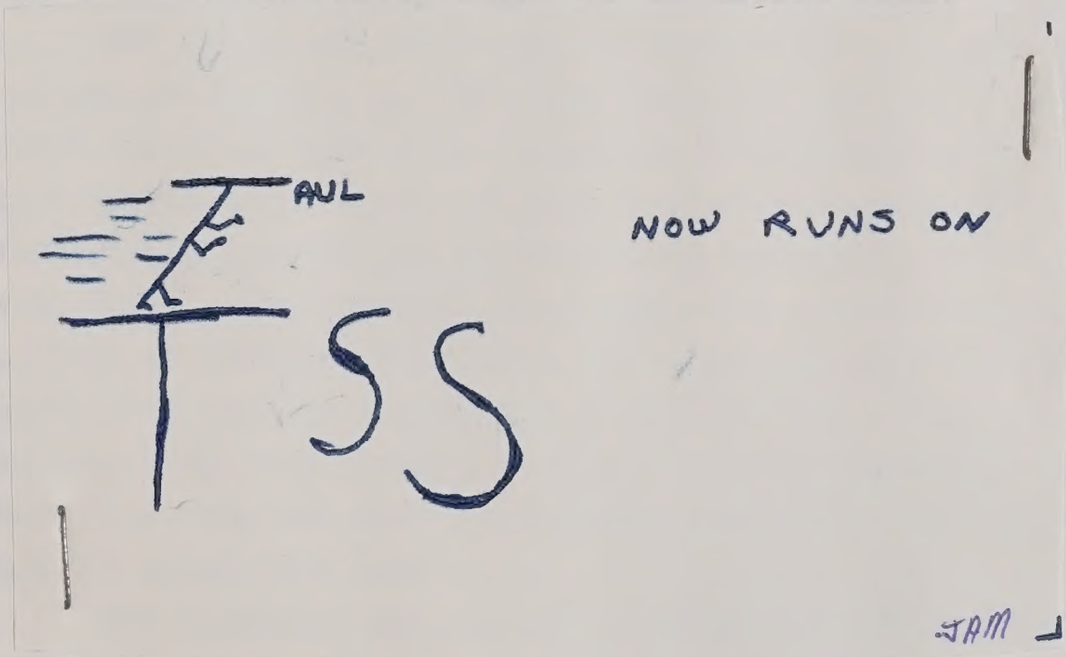
PERCENT.COMPLETE -

is followed by an integer between 0 and 100, and then an end of paragraph (a special character) which can be represented as a delimiter by using the word:

PARAGRAPH

So to get a list of these numbers from a given list BUGS of appropriate subfiles, the command would be

GET TO CENTIPEDE BUGS FROM ==PERCENT.COMPLETE -==,
PARAGRAPH ALL.,



Note that the other string operators work on strings which have been isolated as 1-item lists, or as the top item of a list. GET TO, on the other hand, is a string operator that hops from item to item, going down a whole list consisting of file or string pointers.

GET TO is really the basic way of getting text out of the AIDS files, once the appropriate files or subfiles have been collected onto lists of file pointers. (Such collection is usually to be done by means of SET ALL IF commands and logical operations; see the general discussion of AIDS files for an explanation of this procedure, pp. . Note there also the importance of not calling in catastrophically long strings.) Once the right lists exist, the appropriate words or figures are found in them by GET TO-- which really ought to be called FIND.

GET TO is an important embodiment of the basic design philosophy of TAUL and the AIDS system: that of formatting text files in well-defined ways, so that particular kinds of information will always be found in a particular well-defined surrounding. With the files structured in this way, the corresponding pieces of information for a whole series of files may be pulled out by naming the surroundings. (The alternative philosophy, that of creating pointers into the file, rather than formatting the text exactly, will not be discussed here.)

The surroundings may be further specified according to other things that appear in the text (the STRINGNs).

COMPLEXITIES OF 'GET TO.' There are two particular complexities that affect the use of the GET TO. The first concerns the method of scan, and its consequences for the use of repeated or identical delimiters; the second concerns the varying length of its search memory, which makes it treacherous to look for long strings.

1. Method of scan and the selection of delimiters.

No 9: GET TO starts at the beginning of text and searches rightward. It looks first for a left delimiter. Upon finding this delimiter, it continues searching rightward-- but this time in a new state, which we may call "hot on the scent." It is now looking for what else must be found to satisfy the specified search-- the right delimiter, if no contained material has been specified, or the contained material, if any, has been specified. However, if it finds an identical left delimiter along the way, it will forget the old. Thus, whatever the left delimiter, its nearest instance to otherwise satisfactory text will be selected.

Upon finding the right delimiter, it then makes a string of all the found material. If more strings are to be sought (i.e., if QUANTITY has not been met yet), it continues scanning from a point just after the right delimiter just found.

This means that identical strings may be given as left and right delimiters (since it will recognize a right delimiter before it recognizes a new left delimiter). However, since after a hit it begins scanning to the right of the most recent right delimiter, identical left and right delimiters will result in an alternating, leapfrog pattern of strings returned:

DELIMITER ... DELIMITER ... DELIMITER ... DELIMITER ... DELIMITER

2. Varying length of search memory.

If a subfile specified on LIST2 is more than one record long, it will be searched differently from a subfile only one record long. This is a glitch.

If the subfile is one record long, the delimiters may be "any distance" apart-- that is, at opposite ends of the record. If the left delimiter is at the beginning of the record and the right delimiter does not come till the very end, the resulting string will be the entire record.

This changes dramatically if the subfile is on more than one record. If a left delimiter is on one record and the right delimiter is on a succeeding record, then the intervening text may not be more than 150 characters long. It "forgets" a delimiter that occurred more than 150 characters ago, if that happened on another record, and leaves the "on the scent" state.

However, if you are lucky enough to have the left and right delimiters fall on the same record, then it will give back the entire string between, just as if the subfile were only one record long.

NUMERICAL OPERATIONS

Two numerical operations, addition and subtraction, are available to the TAUL programmer.

The format is:

LIST = LON1 ± LON2 .,

LIST may be any list. If it already exists, its contents are clobbered and lost; if it does not exist it is created. LON1 and LON2 may be a List Or Number. If either is a list, its top element must be a number, otherwise an error will result. The numbers specified by LON1 and LON2 may be positive or negative, but they must be integers, otherwise an error will result. ± must be either plus or minus, but cannot be both.

If ± is +, LON2 is added to LON1 and their sum is stored in the only element of LIST. If ± is -, LON2 is subtracted from LON1 and their difference is stored in the only element of LIST. The contents of LON1 and LON2 are unaffected.

Numbers are stored in decimal as list items. Except when using these numerical operations, they are seen by the processor as no different from any other text. They may be input by the program using the STAR ALPH command.

Examples:

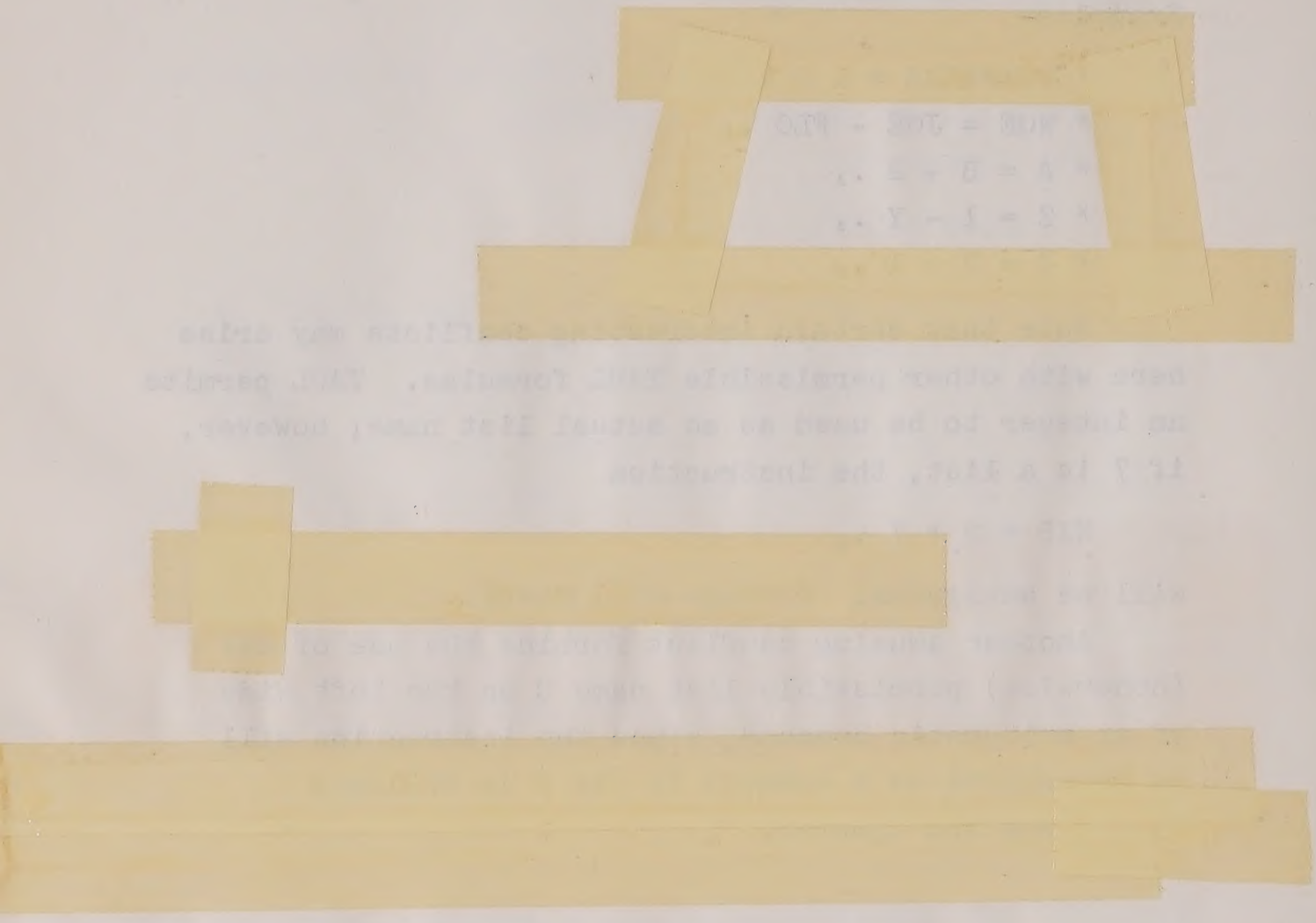
```
* SURPRISE = 1 + 1 .,
* WOE = JOE - FLO .,
* A = B + 2 .,
* Z = 1 - Y .,
* 3 = 2 + 1 .,
```

Note that certain interesting conflicts may arise here with other permissible TAUL formulae. TAUL permits an integer to be used as an actual list name; however, if 7 is a list, the instruction

```
ZIP = 2 + 7 .,
```

will be ambiguous. Garbage will result.

Another amusing conflict forbids the use of the (otherwise) permissible list name C on the left side of an arithmetic command, since the instruction will be recognized as a comment if the C is followed by a blank and ignored.



IF

This command is the only test available in TAUL. It may be used to compare lists, strings, numbers, or particular data from subfiles. If the condition is true the next instruction is executed; if the condition is false, the next instruction is skipped.

The format is

To test whether the lists are equal (and thus skip if they are not equal), the format is

IF DATA1 EQ DATA2 .,

To test whether the lists are not equal (and thus skip if they are equal), the format is

IF DATA1 NE DATA2 .,

The same is true for testing whether DATA1 is greater than (GT) or less than (LT) DATA2.

IF in TAUL is the most weirdly powerful command in the language, perhaps in any computer language. It may be used to compare lists or strings, and if the strings are numbers, treat them numerically rather than as strings. It can also be used to test particular data types from the AIDS subfiles.

In most cases the IF command is used with list names. The lists are compared as lists, or if dissimilar in their tokens, their contents are compared, irrespective of whether the tokens being compared point to files, strings or numerical data. In all these cases the form is

IF LIST1 EQ LIST2 .,

or

IF LIST1 NE LIST2 .,

Examples:

```
IF FURZE EQ GORSE .,
IF LISTON NE CLAY .,
```

The operation of IF upon lists is as follows. First the two lists are compared token for token. If all tokens are identical, obviously both lists are identical. However, if all tokens are not identical, that does not mean the lists are not identical: the material pointed to by one token might be the same as that pointed to by another token. Therefore, the first two dissimilar tokens have their contents-- the matter they point to-- compared. If these contents are identical, the next two dissimilar tokens have their contents compared. This process continues until the IF can be answered-- i.e., until a pair of tokens have been found to have dissimilar contents, or until all pairs of tokens have been found to have identical contents.

Footnote:

When using the GT and LT operators, the ASCII collating sequence is in effect.

The typical use of IF is with GO TO, (see explanation following), so that the first leg of the IF merely forks control to a routine elsewhere:

```
IF (condition) .,
GO TO aaa .,
(more code for the alternative condition)
```

However, there are many other possibilities that do not involve GO TO. For instance, counting occurrences of a condition:

```
IF (condition).,
TICKER = TICKER + 1 .,
(continuation) .,
```

Or the IF may be followed by single commands that take care of the business under a certain condition--i.e., one-command subroutines:

```
IF (condition).,
POP ONE TO SNOOT .,
(continuation).,
```

Note the usefulness but potential mischief of

IF xxx., CALL yyy., zzz.,

If condition xxx is true, procedure yyy will be called. However, procedures always go back to the places where they were called (see procedure commands, below). Hence zzz will follow procedure yyy. In other words, in this example, zzz will occur regardless, but yyy will occur if the condition is true. Thus it is unlike following an IF with a GO TO, which will not return control to the second leg of the IF unless specifically directed to do so. It is instead like the "one-command subroutine" mentioned in the last paragraph.

Note also that a branch whose label is not known to the system is harmless provided the path to that branch is not taken. (The processor does not check the legality of bypassed instructions.)

Examples (where UNKNOWN is the unknown label or procedure):

IF TWEEDLE EQ BZ., GO TO UNKNOWN., GO TO DUM.,
IF ZORCH EQ ZILCH., GO TO ZAPP., CALL UNKNOWN.,

Either of these cases is acceptable to TAUL as long as the unknown branch is not taken.

GO TO

This command transfers the program.

The format is

GO TO LABEL .,

Upon reading this instruction, the system jumps, taking its next instruction from the command having the label named, and goes on from there.

LABEL is any label. Any number of GO TO commands may branch to the same label. The same label may not appear more than once in any program, although it may be referenced by any number of GO TOs.

(To carry out this instruction, the processor looks up the location of that label in the label table, and goes to the instruction address given with it.)

Examples:

```
GO TO TESTAREA .,
GO TO A0100 .,
GO TO 15 .,
SUMIT:  GO TO G09 .,
```

The GO TO is unconditional, containing no tests. However, it is commonly used with a test, i.e., the IF instruction. (See IF, p.).

Forward branching is always forbidden in TAUL except within Procedures. A label to which a GO TO is directed must already exist and have a defined location. (I.e., either before the GO TO, or after the GO TO with the instruction on a leg of an IF that is not taken till the label is in the machine.)

However, if the routine branched to is still being read in from a source (such as cards), execution will take place at the rate in which executable instructions arrive. (Note, though, that there is no convenient way to assure that just this small degree of forward pointing will occur.)

Note that a GO TO which is bypassed and not used-- i.e., the unused leg of an IF-- may reference an unknown or nonexistent label. (This is described under IF, above.)

It is impossible to do a GO TO either into a procedure or out of it. (A procedure must be invoked by a CALL instruction and dismissed with a LEAVE or LEAVE*.)

The label addressed by a GO TO must already be in core, or an error message will result (see "Program Organization," p.).

THE PROCEDURE COMMANDS

The procedure commands are a system of commands for creating TAUL subprograms. These subprograms, or procedures, may be called and combined in various ways. TAUL procedures may (and should) be used to set off the principal program sections; they may be called as subroutines by any part of the program; and they may even call each other or themselves in the most mathematically elegant way.

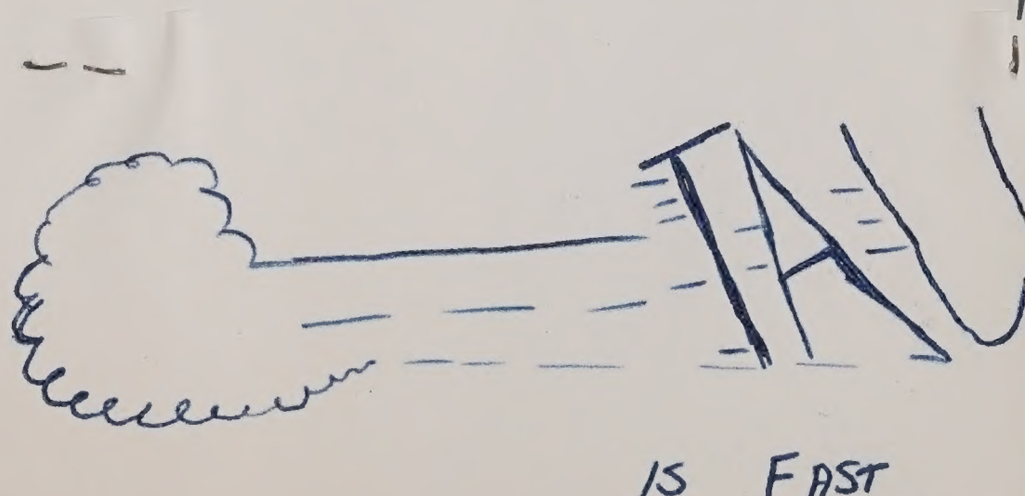
The different procedure commands define the beginnings and ends of procedures, and other parts of departure from them; call the procedures from elsewhere in the program; and even permit the changing of variable-names as procedures are entered and left.

TAUL permits you to be in many procedures at once, calling one other successively, calling themselves recursively, and calling each other back and forth co-recursively.

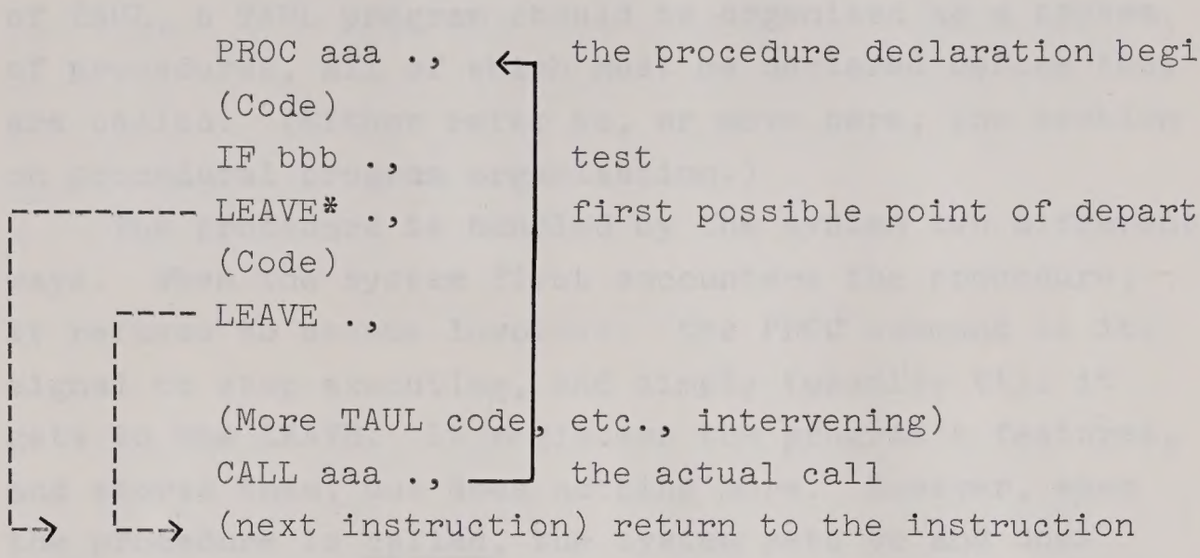
Several different instructions knit together to make a procedure. With this in mind, we will discuss the anatomy of a procedure. After the different parts and their relations have been pointed out the exact features of the individual procedure commands will be detailed.

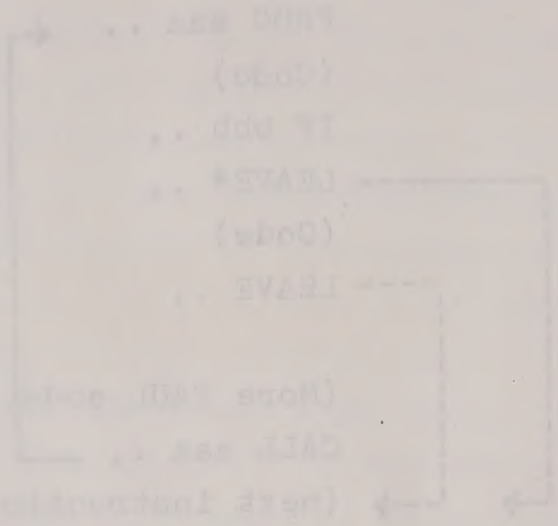
A procedure is defined by a PROC instruction, which means that the procedure is about to be described-- not executed. Then came the commands that are to take place in the procedure. Finally, a LEAVE instruction signals the end of the procedure, though other instructions (LEAVE*) mark alternative exit points along the way. Finally, elsewhere in the program, the procedure is summoned by a CALL instruction. Then and only then is its code executed.

One minor complication-- voluntary, of course-- is that data names may be deliberately changed during a procedure. This is not just to protect the innocent, but also to permit procedures to be separately coded and debugged without regard to the programs which will use them later.



The typical anatomy of a TAUL procedure looks something like this:


PROC aaa .,
the procedure declaration begins.
(Code)
IF bbb .,
test
LEAVE* .,
first possible point of departure
(Code)
LEAVE .,
(More TAUL code, etc., intervening)
CALL aaa .,
the actual call
(next instruction)
return to the instruction right after the CALL presently being serviced. The procedure will always pass control back to the place it got it.



The procedure commands have a drastic effect on program organization. Owing to some curious features of TAUL, a TAUL program should be organized as a system of procedures, all of which must be declared before they are called. (Either refer to, or move here, the section on procedural program organization.)

The procedure is handled by the system two different ways. When the system first encounters the procedure, it refuses to become involved: the PROC command is its signal to stop executing, and simply tokenize till it gets to the LEAVE. It registers the program's features, and stores them, but does nothing more. However, when the procedure is called, the system sets to and does what it has merely duly noted before. The one advantage is that forward branching is possible within a routine.

With this general exposition behind us, let us look at the individual procedure commands. After the procedure commands are detailed formally, which will probably be unclear at first reading, they will be discussed informally.

PROC

This command indicates the beginning of a procedure, and tells what the name of that procedure is going to be. (In its more elaborate form it also serves as table top on which to put variables being passed into the procedure.)

The format is

PROC PROCNAME (LIST1 , LISTN) .;

wf

PROCNAME is the name of the procedure. LIST1 and the LISTNs, all of which are optional, are dummy names for lists which will be brought in later. Wherever these dummy list names are used in a particular procedure, incoming list names will be substituted from the CALL instruction which summons it.

When called (by a CALL command), the PROC instruction is the point from which execution continues, until a LEAVE or LEAVE* command is encountered by the processor.

When encountered, the PROC command causes the processor to file the procedure name and the location of its first command (that which follows the PROC, unless it is a comment, which is not stored). The PROC instruction also switches off an Execution Flag; execution ceases until the processor encounters a LEAVE command.

The total number of procedures permitted within a program is presently restricted to ten (10). The number of procedure calls is, of course, not restricted, except by the usual limitations of program size and (for successive calls) the virtually unpredictable limitations of list space. Various lists have to be maintained for procedure calls, and in the case of successive and recursive calls these might grow to considerable size. If a procedure calls itself many times without a LEAVE, and each time passes itself several arguments, space may well run out.

DECLARATION OF PROCEDURES. For reasons discussed elsewhere ( ), it is recommended that all procedures be declared at or near the beginning of a program, though this is simply a convention intended to help you avoid trouble.

A procedure may not be declared within a procedure. Thus all procedures ought to be declared before any procedure calls occur. (Note that this simply follows from the advice in the above paragraph).

Two procedures may not have the same name in the same program.

The simpler case of the PROC command changes no names, passes no arguments. Its form is simply

PROC PROCNAME .,

This signifies the beginning of a procedure. The procedure, when called, is executed upon data exactly as specified. Naturally, a procedure can operate upon any data known to the program.

Sample simple procedures:

```
PROC SCHNITZEL.,
ABERNATHY = ABERNATHY -1.,
GLOCKENSPIEL = GLOCKENSPIEL -2.,
LEAVE .,

PROC GLITCH.,
*L1 = SUB (L1, 2).,
INDEX = INDEX + 1.,
*CHECKER = LENGTH (L2).,
IF INDEX NE CHECKER.,
CALL GLITCH.,
SET INDEX = ALPH (0).,
LEAVE.,
```

The PROC command also has a provision for changing the names of data while it is being used by procedures. In this more complex case of the PROC command, variables are "passed" to the procedure. You are supposed to visualize this as a way of handing over particular variables to a routine: a sort of counter-top on which they are placed.

The format is the full PROC format:

PROC PROCNAME (LIST1 , LISTN) .,

LIST1 and the LISTNs are variables to be "passed" to the procedure by the CALL command. CALL must specify the same number of lists as the PROC command it calls. The lists supplied by the CALL command are plugged into the PROC command. Effectively, this means that the names of the lists in the CALL are turned into the names of the lists in the PROC for the duration of the procedure.

Example:

```
PROC NECESSITY (L1, L2).,
(Code affecting L1 & L2)
LEAVE.,
  .
  .
  .
CALL NECESSITY (MOTHER, INVENTION).,
```

The list called MOTHER becomes renamed L1 while the procedure is in effect, and INVENTION becomes renamed L2 while the procedure is in effect.

This is about equivalent to:

```
PROC NECESSITY.,
:
(Code affecting L1 and L2)
LEAVE.,
:
SET L1 = MOTHER.,
SET L2 = INVENTION.,
CALL NECESSITY.,
SET MOTHER = L1.,
SET INVENTION = L2.,
```

You will note that passing variables through the PROC and CALL saves some trouble. It is particularly useful where the same thing will be done to data on different lists, and you'd rather not create a buffer list (or lists) to carry operands in and out of the procedure.

Sample procedure with "passed variables":

```
PROC FOODLE (GOD, MOTHER, COUNTRY).,
OOPS..POP MOTHER TO GOD.,
COUNTRY = COUNTRY -1.,
IF COUNTRY NE NULL.,
GO TO OOPS.,
LEAVE.,
```

The way this is actually done is as follows (--if the terms are unclear, see the section on TAUL inner mechanisms, pp. ; if more detail is desired, see the section on procedure handling, pp.): the procedure exists as a list of program tokens.

The list names given in the PROC command are marked as dummies. At run time, these list names are replaced with the list names given in the CALL. (Actually, the program tokens pointing to these list names are replaced with tokens pointing to the list names supplied by the CALL). Upon hitting LEAVE or LEAVE*, the processor undoes this, replacing the list-name tokens supplied by CALL with the dummy list-name tokens that were there before.

Often a procedure can be called again before its operations are completed. This occurs when the procedure calls itself, or calls another procedure which then calls it back. In this case the list names that have been changed must be changed again; but provision must exist for them to be changed back. This is done by maintaining, for each procedure, a stack of lists whose names have been replaced in the procedure. Each time the procedure is called before completion, the present changeable list-names go on top of the stack; each time that procedure hits a LEAVE instruction, the most recent additions to the stack go back into the procedure. On the final LEAVE, the original dummy variables named in the PROC, which were at the bottom of the stack all along, are plugged back into the procedure, the stack is empty, and all is as before.

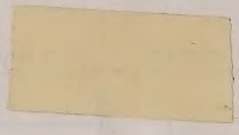
Thus procedures may call each other and "pass variables" ad lib. But list space is finite, so that, say, a program a hundred calls deep in recursion, passing several variables, might bomb. Such a program might have a routine like

```
PROC GREMLIN (L1, L2, L3).,
POP L1 to L3.,
POP L2 to L1.,
POP L3 to L2.,
COUNTER = COUNTER +1.,
IF COUNTER EQ PRESET.,
CALL GREMLIN (L3, L1, L2).,
LEAVE.,
```

CALL

This is the command which, when encountered, causes the execution of a procedure declared elsewhere.

The format is:

CALL PROCNAME (LIST1 .,LISTN) .,
↑____|wf

The procedure named is executed at once. On termination of the procedure (LEAVE* or LEAVE), program execution will jump back and continue from the particular CALL command. The lists named, if any, have their names temporarily substituted for the list names given in the PROC instruction of the procedure being called. (For fuller explanation, see this process explained as "passing variables" under PROC (above), or the more extensive treatment of mechanisms (either below, in this section, or later in the section on TAUL mechanisms).)